The Cassini Spacecraft: Object Oriented Flight Control Software

John C. Hackney, Douglas E. Bernard, Robert D. Rasmussen

Jet Propulsion Laboratory, California Institute of Technology
Pasadena, California

Abstract

The Cassini mission will explore the Saturnian system In much greater depth than was possible by the Voyager fl yby missions. The spacecraft is comprised of a Titan pmbc and a Saturn orbiter.

The Cassini Attitude and Articulation Control Subsystem (AACS) is responsible for determining and cent rolling the spacecraft at titude including instrument pointing, antenna pointing, and thrust vector pointing during velocity change maneuvers. The 12 year mission life, long round- trip light time, and extended periods of coast without continuous ground control drive the AACS flight software design in the directions of autonomy, fault tolerance, and modularity to accomodatc planned upgrades in flight.

Past experience on JPL planetary programs indicated the need for a fresh approach to specifying, and developing AACS flight software. An object oriented appr each offers many attrac.live advantages over more conventional methodologies. These include an improved m od - ularity in both code and data that simplifies program structure, an emphasis on earlier specification and architectural design of software modules, and the ability to carry the design paradigm more directly into flight software implementation.

Recognizing these advantages, the Cassini AACS team has cooperatively tailored a successful object oriented methodology that aids the development of requirements, fosters the earl y consideration of practical implementation issues, and provides a convenient, self- contained vehicle for the delivery of algorithms to flight software.

The Cassini AACS Flight Software is depicted in increasing levels of detail using a Context Diagram, Architecture Diagrams (i.e., Dependency Diagrams), an object Diagram for each object, and a Statechart (i.e., State Transition Diagram) for each object. The detail contained in the diagrams is enhanced and refined during the Requirements and Design Phases of both Subsystem and Software Development. Examples of al 1 the diagrams as well as the criteria for object selection, the advantages of statecharts, and the case of modi - fying the design to accomodate changes in scope ar c described.

INTRODUCTION

in 1989, NASA initiated the CRAF and Cassini missions. These were to have been the first in a series of interplanetary missions for the Mariner M ark 11 program. CRAF was a comet rendezvous mission;

Cassini was targeted for an orbital tour of Saturn; and both were planned to encounter an asteroid along the way. Later missions in the Mariner Mark II series were to be of similar scope, with a goal to study several other objectives in the solar system much more closely than any previous exploration.

The CRAF and Cassini spacecraft were very large. Both missions imposed huge propulsion demands; each carried numerous instruments; and many types of experiments were planned. Scientifically, the two missions were strikingly different, with little in common between their payloads. Nevertheless, to complement their scientific prowess, the Mariner Mark II spacecraft were also to share many high performance design features, with most of the core engineering subsystems identical, or easily configurable in a modular way. This made the task doubly complex - to meet two unprecedented sets of demands, but in a common design (with room to accommodate needs of future missions in the bargain).

At the core of this complexity was the flight software - that free- flowing but notoriously unwieldy well spring of functionality. With this daunting situation at hand, and knowing the extraordinary effort that had been required to deliver capable and reliable flight software on prior, *much* simpler interplanetary missions, the guidance and control team made an early decision to meet this new challenge with a new approach.

After intense but fruitful debate, a formal approach was launched in mid 1990 which has guided the software design through preliminary development. This approach uses, as its fundamental basis, object oriented methods and statecharts (Ref. 1), but also involves the extension and integration of these methodologies outside the software domain among other areas within guidance and control.

We are happy to report that our experience with this approach has proven to be remarkably effective. Moreover, our ability to regroup with relative ease through the bounding evolution of the program to its current state is owed in some measure to the success of this approach. This paper describes the motive for our method, the nature of its essential constituents, and its application from initial conception through subsequent evolution of the project. To set the stage we begin with a brief description of the CRAF and Cassini missions and spacecraft as they were originally conceived when this approach was developed.

THE CASSINI CHALLENGE

The Cassini mission will explore the Saturnian system in much greater detail than was possible by the Voyager missions. Both Voyager 1 and Voyager 2 flew by Saturn in the course of their tour of the outer solar system, but never went into orbit. The Cassini mission plan calls for four years of intensive study of the Saturnian system following Saturn orbit insertion.

Mission Background

The Cassini Flight Software approach has already been tested by programmatic and spacecraft design changes. Both the CRAF and Cassini spacecraft needed the capability for full three- degree- of-freedom orientation control to support antenna pointing and propulsive maneuvers, and to protect certain spacecraft and instrument surfaces from excessive direct solar radiation. Both spacecraft boasted two- degree- of- freedom articulated High Precision Scan Platforms (HPSPs). CRAF (see Fig. 1) had an additional limited- motion, low precision, one- degree- of- freedom scan platform; while Cassini (see Fig. 2) had a low- precision, one- degree- of- freedom probe relay antenna and a continuous - rotation turntable. The Imaging Subsystem (ISS) cameras on the HPSP were time-shared between imaging science uses and star detection and location to determine the "ISI"- and

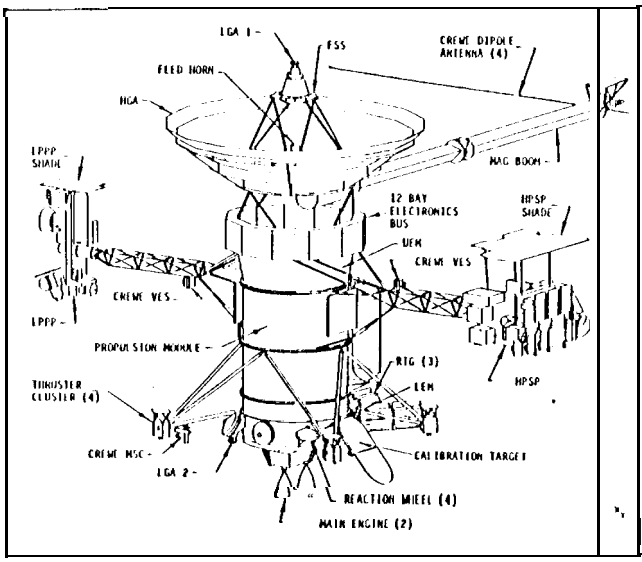

Fig. 1 CRAF 1991 Configuration

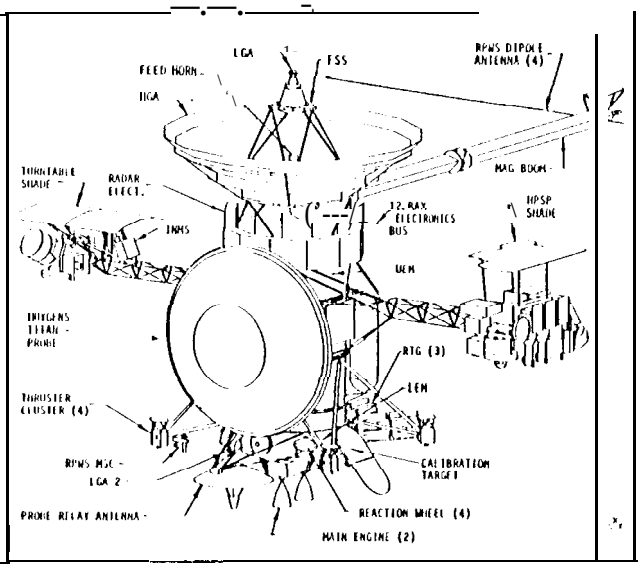

Fig. 2 Cassini 1991 Configuration

thereby the spacecraft attitude. The CRAF/Cassini AACS software architecture discussed below accommodated these and other software tasks in a structure that, with the exception of turntable control, was identical for CRAF and Cassini.

In January, 1992, NASA's final budget did not include CRAF, and the same budgetary pressures that eliminated CRAF were challenging Cassini's survival in its then-current form. After reexamination of mission goals and requirements and spacecraft options, a simplified redesigned Cassini spacecraft was proposed which could be delivered at lower cost and cost risk than the earlier baseline. The redesigned Cassini has since been adopted as the Cassini baseline and will be discussed in more detail below.

Mission Objectives

The spacecraft is comprised of a probe which will take atmospheric measurements during its descent to the surface of Titan, and a Saturn orbiter which will investigate the satellites, rings, atmosphere, and magnetosphere of Saturn over the course of 4 years and 60 Saturn orbits. During this mission, Titan will come in for particular scrutiny with 33 flybys for imaging, RADAR, and radio science observations.

The control software under discussion is responsible for determining and controlling the Cassini Spacecraft attitude at all times in the mission including camera and instrument pointing, probe pointing at probe release, antenna pointing for communications, probe data relay, RADAR, and radio science, and thrust vector pointing during velocity change maneuvers. The 12 year mission life, long round-trip light time, and extended periods of coast without continuous ground control drive the AACS flight software design in the directions of autonomy, fault tolerance, and modularity to accommodate planned upgrades in flight.

CASSINI SPACECRAFT

Fig. 3 shows the redesigned Cassini spacecraft. The most visible changes from the earlier Cassini design are that the platforms were removed and all instruments and antennas became body fixed. This meant that all instrument, sensor, or antenna pointing would need to be accomplished by spacecraft reorientation. Since spacecraft reorientation is relatively slow, it was no longer feasible to time-

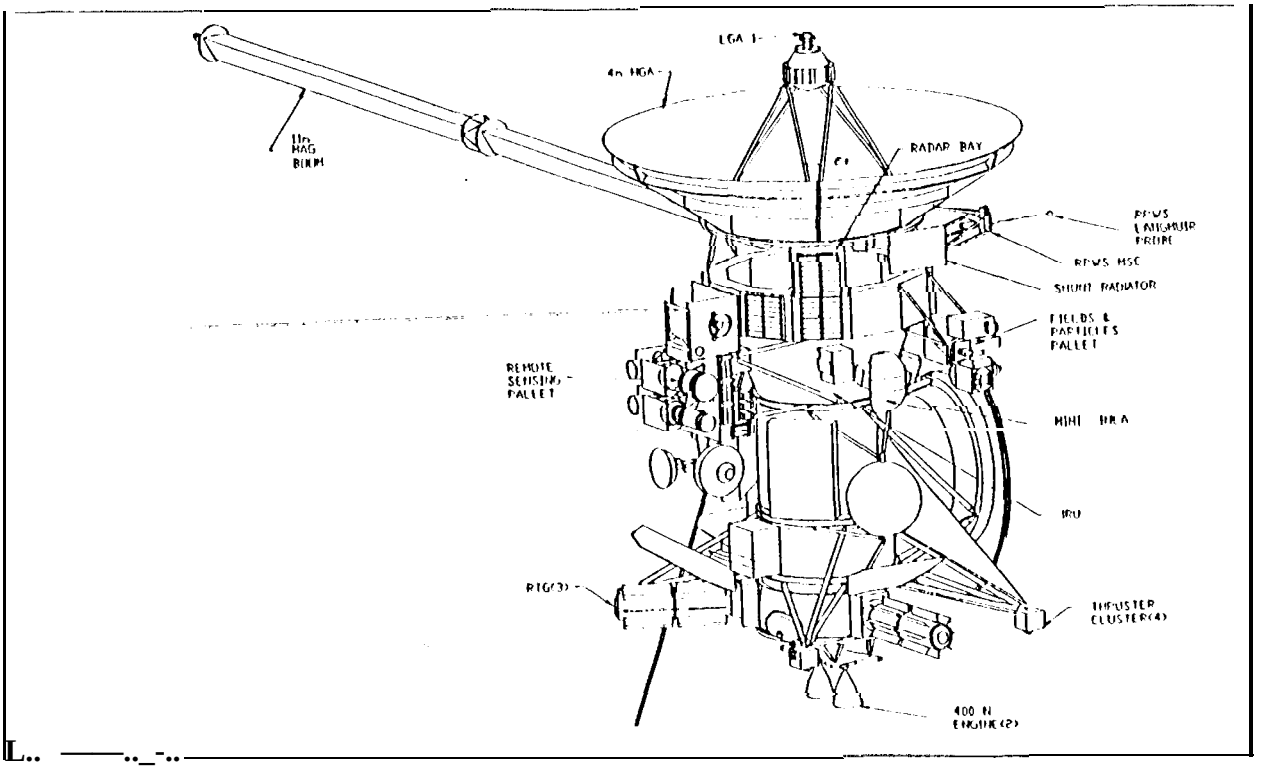

Fig. 3 Cassini 1992. Baseline

share the 1SS cameras between imaging and AACS. Instead, a dedicated star tracker was added which]s capable of continuous star viewing.

As before, Cassini carries the large (3 m diameter) **1** luygens Probe for ejection into the Titan Atmosphere.

Cassini now features a remote sensing pallet which includes all precision optical instruments. The optical experiments are co- boresighted so that aiming one of them points them all at the same target. These instruments all have radiators mounted perpendicular to their boresights to cool their detectors. 'l'he radiators must be kept out of direct sunlight at all times of the mission for the instruments to function properly. 'l"he Stellar Reference Unit (SRU) used for star tracking is also on the pallet. Its boresight is pointed in the same direction as the science instrument radiators so that the SRU field of view will never be obstructed by either the sun or the optical instrument's target.

Other fields and particles instruments are mounted elsewhere on the spacecraft and require differ ent spacecraft at tit udes and motions to collect their data, Onc group of experiments requires the spacecraft to roll about the axis of the High Gain Antenna (1 IGA) for up to 8 hours so that the experiments can take measurements in all directions.

The 4 m wide} Iigh Gain Antenna serves many purposes. It is pointed at the Earth for communications between Cassini and the mission operations, The 1 IGA will track the motion of the probe as it enters '1 'itan's atmosphere and receive data beamed back from the probe for later ret ransmission to the 1 Earth. It serves as a radar transmit ter and receiver for 'l'it an alt imetry and mapping experiments. Finally, the IIGA may be used for Radio science experiments ranging from gravity wave detection to atmospheric or ring particle studies during Earth occultation by Saturn, its rings, or Titan.

in order to generate adequate power at over 9 AU from the sun, Cassini is powered by three Radioisotope Thermoelectric Generators.

The propulsion system offers a (redundant) gimbaled bipropellent main engine for high efficiency during over 100 large propulsive maneuvers and small fixed monopropellent thrusters for precise small propulsive maneuvers and three axis attitude control. These small thrusters and the equipment supporting them are collectively referred to as the reaction control system (RCS).

Both the Command and Data Subsystem (CDS) and the Attitude and Articulation Control Subsystem are built around a pair of 1750A microprocessors programmed in Ada. The availability of a language like Ada for the flight computer raises the desirability of choosing an object- oriented software development approach. AACS will be discussed in more detail in the following section,

Commands relating to AACS are inserted into a command sequence, received at the 1 IGA (or one of two low gain antennas), decoded, and sent to CDS. CDS then installs and activates the sequence and sends commands to all subsystems- - including AACS- at the time indicated in the sequence. Commands to AACS are sent out over a 1553B bus to AACS.

Thermal control of the spacecraft is managed by designing certain sides of the spacecraft to accommodate full solar input at 0.61 AU from the sun and other sides (such as the side containing the instrument radiators) which are designed to never see the sun. When close to the sun, the 1 IGA is generally sun- pointed. All turns off sun point at this phase of the mission are brief and in a direction so that the sun is on the probe side of the spacecraft. Thermal constraints form the basis of some of the requirements for fault protection software speed of response.

ATTITUDE AND ARTICULATION CONTROL SUBSYSTEM

Functions

The Attitude and Articulation Control Subsystem is responsible for attitude determination, attitude control, trajectory change maneuvers, fault protection, spacecraft attitude broadcast, and telemetry.

- Attitude Determination: determination of the sun location using the Sun Sensor; star identification using the Stellar Reference Unit (SRU); attitude determination using identified stare, and attitude propagation between star updates using a dynamic model augmented at times by the Inertial Reference Unit (IRU).
- Attitude Control: Control of the spacecraft attitude using thrusters for thermal control, HGA pointing, probe pointing, and small trajectory change maneuvers; control of the spacecraft using engine gimbal actuators augmented by roll control thrusters during large (main engine) trajectory change maneuvers; control of the spacecraft attitude using reaction wheels (RWAs) during most of the Saturn encounter period.
- Trajectory Change maneuver magnitude control: burn termination based on timers for small maneuvers, based on an accelerometer backed up by a timer for large maneuvers.
- Fault Protection: Failure detection, location and recovery for failures of attitude control and propulsion assemblies.
- Attitude Broadcast: The spacecraft attitude and angular velocity with respect to the Inertial J2000 coordinates system is broadcast to all processors on the spacecraft that may need to know it

- Telemetry: Telemetry is designed to allow ground reconstruction of spacecraft attitude, AACS assembly health, and nominal and fault related AACS activities.

Cent rol 1 oops

AACS is functionally represented by Fig. 4. This figure shows that the AACS Flight Compute]

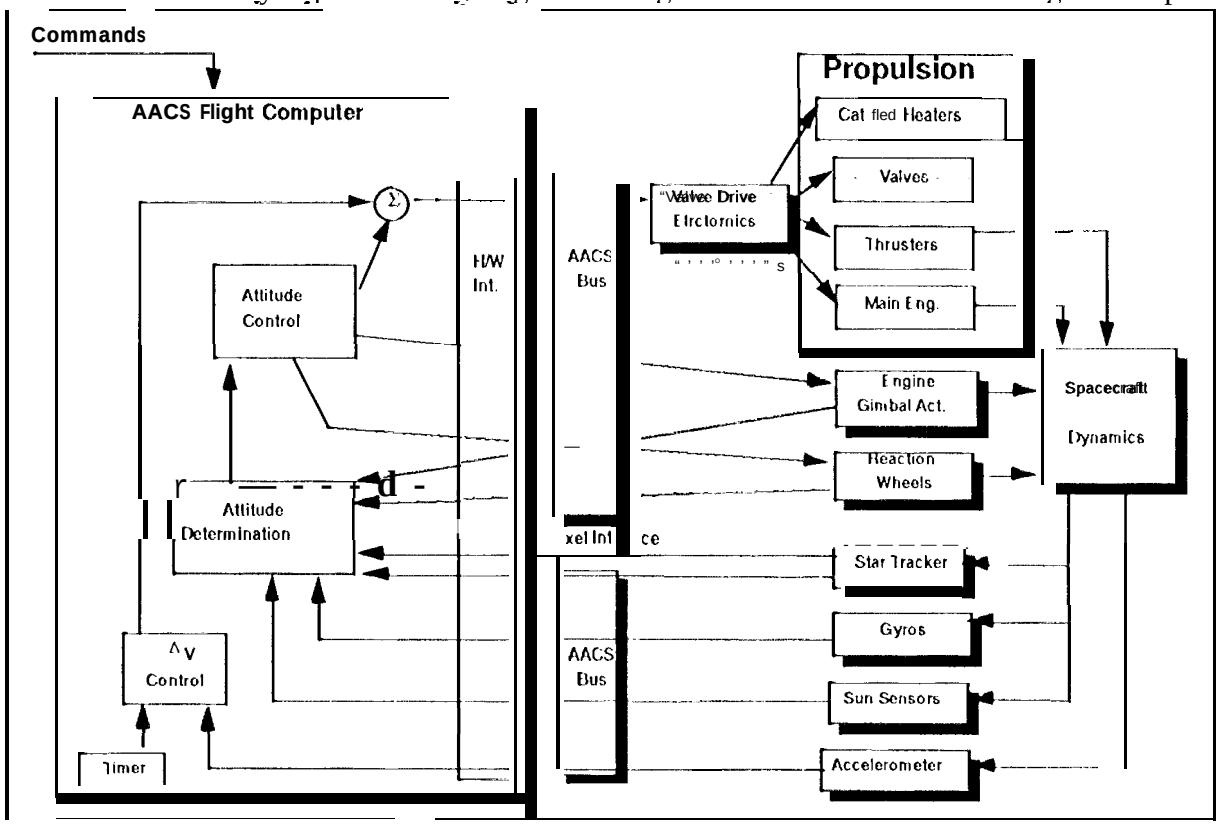

Fig. 4 AACS Functional Block Diagram

(AFC) communicates with CDS and a large number of AACS peripheral sensors and actuators. 'l'he sensors include a redundant Inertial Reference Unit , Stellar Reference Unit, and Sun Sensor, and a non- redundant Accelerometer. Actuators include redundant reaction wheel assemblies and Engine Gimbal Actuators. The AACS also controls Propulsion Subsystem valves, thrusters, engines, and heaters. Except for the SRU, all communication between the AFC and the AACS peripherals are via the AACS bus. The SRU sends its voluminous CCD image data to the AFC via a dedicated pixel interface.

Figure 4.1 captures the basic contr o] loops. Note that the "blocks" discussed here will form the basis for some of the objects selected below.

in the attitude control loop, commands from CDS drive an attitude commander which determines the instantaneous attitude, angular velocity (attitude rate), and angular acceleration commands. Data from the sun sensor and the IRU is combined to generate an attitude and attitude rate estimate in the attitude determination block. This is passed to the attitude control block which compares the commanded and estimated attitude and attitude rate, notes any commanded angular acceleration, and, depending **on** cent rol mode, generates commands for the appropriate combinat ion of thrusters, reaction wheels, or EGAs.

In the velocity change control loop, a timer is used to terminate the propulsive maneuver when the desired velocity change has been achieved. For main engine maneuvers, the timer is augmented by an accelerometer which measures one component of the velocity change directly.

AACS Modes

Another view of the AACS is provided by Fig. 5, "AACS Flight Software Modes". This figure

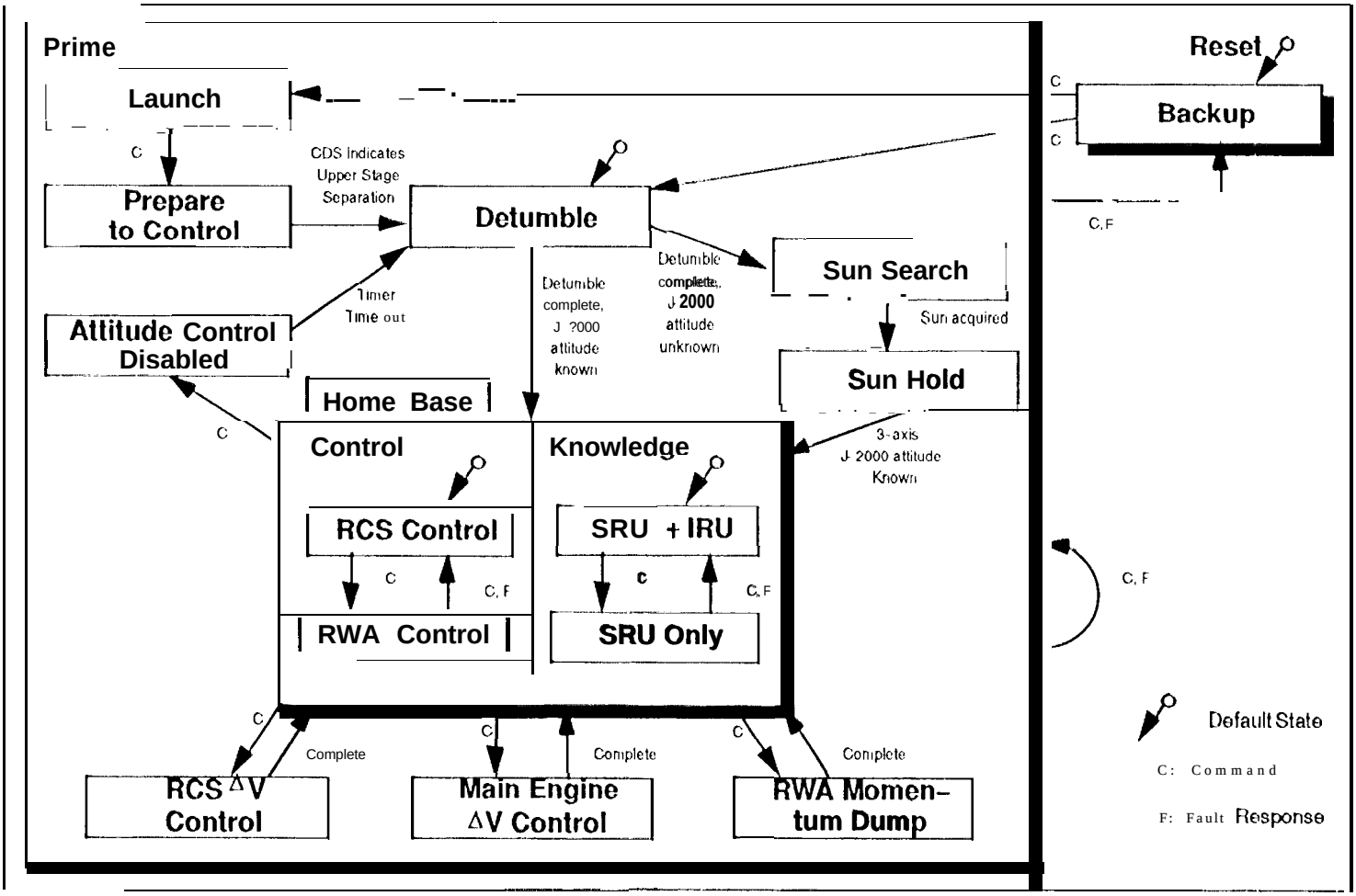

Fig. 5 AACS Flight Software Modes

shows the major subsystem modes. The Flight Software in an AFC will be in one and only one mode at any one time. No more than one AFC is allowed to be prime at a time, so if one AFC is in the Prime supermode, the other AFC is either off or in Backup mode. The figure should be read as follows: The symbol with an arrow drawn from a dot denotes the default state or substate. When a state is entered without a positive indication of which substate to start at, the FSW starts at the default state. The letter C refers to a commanded state transition; AACS receives these commands from CDS. the letter F indicates a transition caused by an internal AACS fault response. If a state is divided by a vertical line (see Home Base), that means that there are two concurrent machines and the total state is made up of the combination of the active substate on each half of the vertical line. The modes are discussed below by considering a couple of scenarios which exercise most of the transitions shown.

Before launch both AFCs are powered, and they both come up in the Backup mode. One AFC is then commanded to become Prime in the Launch mode. in the Launch mode, some hardware can be powered for self protection against the launch environment, but no spacecraft cent rol is attempted and it is not possible to fire any thrusters or main engines. Following burnout of the upper stage, but before upper stage jettison, AACS is commanded into Prepare to Control mode. Here the propulsion subs ystem can be vented and valves opened and closed as necessary t o be read y t o f ire thrusters. in addition, the Attitude determination function is initiated using the sun sensor and IRU.

As soon as AACS receives an indication from CDS that the upper stage has been jettisoned, it transitions to Detumble mode. In this mode, the only objective is to achieve a target spat.eclaft rate c1osc to zero. At this point, the inertial (J2000) attitude is unknown and a transition is made to sun search mode. in the nominal case, the sun is centered in the sun sensor FOV at upper stage jett i son, so the spacecraft has enough information to give deterministic commands which will turn the sun sensor back to directly reacquire the sun without the need for a full sky search. If the sun location were completely unknown, such as after an attitude knowledge fault, a full sky search would be initialed that would be guaranteed to find the sun within a predetermined amount of time.

Once the sun is acquired or reacquired, a transition is made to sun hold mode which keeps the sun sensor pointed at the sun and keeps the drift rate about the sun line below a certain allowable value.

At a later time, the SRU will be turned on, checked out, and the software will identify the stars in its field of view. This will allow a full 3- axis determination of the spacecraft attitude with respect to the J2000 inertial coordinate system.

At this time, the software will make its first transition into the 1 Jomc Base mode. Except for brief planned (and possibly unplanned fault response) excursions out of Home Base, the software will spend the rest of the mission in this mode.

} Iome Base has concurrent Control and Knowledge substates. That is, whenever the mode is I lomc Base, there will be both a Control and a Knowledge submode. The Control submode will be RCS or RWA cent ml, depending on which actuator is used to control the spacecraft. For Knowledge, the SRU will always be used, but the distinction between the submodes depends on whether the IRU is also used to augment the SRU in attitude propagation.

The Probe Release scenario starts with the spacecraft in Home Base mode in a "Cruise" state defined by RCS Control and SRU - only Knowledge. The IRU is needed for Pi obc release, so a mode switch to SRU -J IRU will be commanded.

Once the IRU has warmed up and other preparations have been made, the AACS is commanded into Attitude Control Disabled mode a few seconds before probe release. This is to avoid damaging the probe with combustion products during its departure from the spacecraft, Att itude Control Disabled is a t imed mode, with the time- out t imc as a variable parameter. For Probe release, the timer will be set to expire a few seconds after probe release.

At this t imc, the spacecraft will execute a detumble maneuver, and, given the large kick the probe gives the spacecraft, it will take many minutes to bring the spacecraft angular velocity back close to zero.

Since the IRU is on, AACS never looses track of the spacecraft attitude, so on completion of the detumble, transition is made directl y back to} Iomc Base, using the default substates of RCS Control and SRU + IRU.

With this overview of the AACS functionalit y providing some insight into the diverse and complex requirements placed on the AACS Flight Software, let's turn to how wc went about selecting the methodolog y for flight soft ware develop ment.

SELECTING A METHODOLOGY

1 lad there been strong support for a particular methodology within the guidance and control team at the inception of the project, little debate would have ensued over the approach to bc taken in this task. on the contrary, despite past efforts with various methodologies, the general sense initially seemed to bc one of unenthusiastic resignat ion to adopting the recentl y used methods of another prog r am.

These methods had proven useful, but unsatisfying, with much of the effort driven only by a desire to "do the job right" by following the prescriptions of the method. As a consequence, most part ici - pants never took full ownership of the approach - particularly those whose close contact with the soft ware development effort was by circumstance instead of desire: control analysis, tracking, system design, integration, fault protection, testing, and so on. This was the state of advocacy going in to the planning process.

This preponderance of apathy must bc honest] y accounted, however, as an unsurprising consequence of the teams composition at that time. "l'he conscious effort to tackle this problem at threadiest stages of the project *and* to enlist everyone with a peripheral role in soft ware development resulted in a team with software specialists decidedly in the minority. Finding an approach that would engage this group to the point whew all could work in a well coordinated manner was essential.

These areas had all certainly worked together on past programs, but history had taught a number of hard won lessons. It was gener al practice, for example, to serialize many processes. S ystem designers would put together the basic architecture and requirements. Analysts would refine the design and then produce the appropriate algorithms to bring about the desired behavior. Software engineers would then mold these assorted algorithms into code - not infrequently with a fair amount of distortion. At some time, usually too late in the progression, a fault protection design would bc wedged into place, and the software would then, final] y, be ready for integration with the rest of the system.

Despite a fair amount of obvious recursion in the process, each of these stages would bc scrutinized carefully, and planners would come to the "logical" conclusion that this serial order was inevitable. One need only look at the tools of the trade to see, for example, that the analysts equations and the programmers code were incommensurate! (e.g., different development and test environments, different languages.) As a consequence, algorithms often evidenced little consideration for their ultimate form when they were del ivered for coding.

Finding a way to break this serialization was a major issue. It was not enough that the software engineers issue guideli ncs, or watch over shoulders to prepare themselves for the i nputs they would receive. T 'he approach selected had to bc one in which interactions among software components was clear and explicit from the beginning, and in which algorithms that are typical of guidance and control applications could find natural expression. This led to a number of ideas that eventually took form in the selected approach.

Foremost is the concept of interconnected state machines. These easil y capture both the sequent ial control constructs of the mode logic, configuration control, and other discrete aspects of the software design, as well as the state dynamics of a control law or an estimator, Both fit nicely into a common framework which the economy of statechart representation makes simple to manipulate and under-

stand. one finds in the state diagrams a common vocabulary that permits the software design to become, not just the recipient of the development ideas, but also a capable expression of them. As a result, wc now have many practitioners of this approach outside the software team who find this to bc a powerful tool in their work, even at early conceptual stages.

State machines carry with thcm the notion of strong association among a collection of data. Advancing the state of the machine is a well defined concept in which allowed transitions are carefully and explicitly delineated, and in which, ideally, the internal processes of state evolution are hidden. These notions also describe the basic tenets of object oriented methodolog y which has become a dominant force in the software world in recent years. Although Ada (the language chosen for this work) only weakly supports software objects, this approach nevertheless fits nicely with the paradigms of the language.

Altogether, the blending of all these ideas seems almost ideally suited to guidance and control from concept ion through implementation. There appears to us to bc no other approach as capable of spanning this range of the development cycle.

The explicit nature of interaction among state machines that is cncoul'aged by this method addresses another weakness that has been apparent in past programs and that also is not approached so directl y by most other methods. Foi example, typical practice in the past has often been to communicate among soft ware modules via common pools of data. '1 "his is a passive form of int tract ion whereby act ivit ics in one area are affected by another only by an overt act of the affected area to determine whether a change has occurred in the shared data.

With effort, shared data can bc manipulated to model a more direct interaction, but when several areas react to common data or contribute to common data, there are no ready means of synchronization among thcm other than the exercise of great caution, A large amount of time is spent with such an approach coordinating all of these interactions, particularly the unintended ones which inevitably and craftily creep in. The great fear is that such interactions may go undiscovered.

While it is possible, with a little willfulness, to duplicate these problems with objects, the notion of event- driven actions as the natural expression of an object's behavior tends to dissuade more passive implementations. Event- driven interactions are not only explicit, but also amenable to mole direct synchronization. In considering the resulting network of control that becomes apparent with this approach, a poor partitioning of funct ions among objects can bc i dent ified and corrected.

For example, wc have used a clear hierarchy among objects which directs control predominantl y downward to promote a manageable structure. Lateral actions are limited, and upward actions are strictly one- to- one actions without the attendant problems described above.

1 n cent rast to the level of control over interactions afforded by most other methods, the object oriented approach has clear advantages. Given this and the other factors cited above, the motives for pursuing this approach were clear. It remained onl y to adopt a formal implementat ion of this method. To do that, wc built a prototype.

Wc conducted a Methodology Prototype with the AACS groups mentioned above participating. The goal of the prototype was to work as far as possible through the process, resolving issues as they arose. The result was a groundwork for the Object Oriented approach eventual I y adopted and agreemcnts on the form and types of deli verables to the flight soft ware devel opment process.

AN OBJECT ORIENTED APPROACH

An object is a set of data and any operations which act on that data. Object boundaries are chosen to maximize the cohesion within a given object (make it as independent as feasible) and minimize

the coupling between objects (as few connections as feasible both in number and frequency). Data within an object may only be changed by operations within the object (there is no common area where data is manipulated by more than one object). Such are the basic tenets of an Object Oriented approach.

The approach presented in the following sections was based on the work of David 1 larel (Ref. 1), Sally Shlaer and Steven Mellor (Ref. **2.),** and Don Firesmith (Ref. 3), but is unique to JPL Cassini AA(3 Flight Software. During the Methodology Prototype, wc defined and refined how to usc the various represent at ions and development techniques to suit our needs. Wc f urther refined the process during the development of a second protot ypc where wc produced an act ual working model by going through all the steps in our development process as portrayed in Fig. 6 and using the software devel-

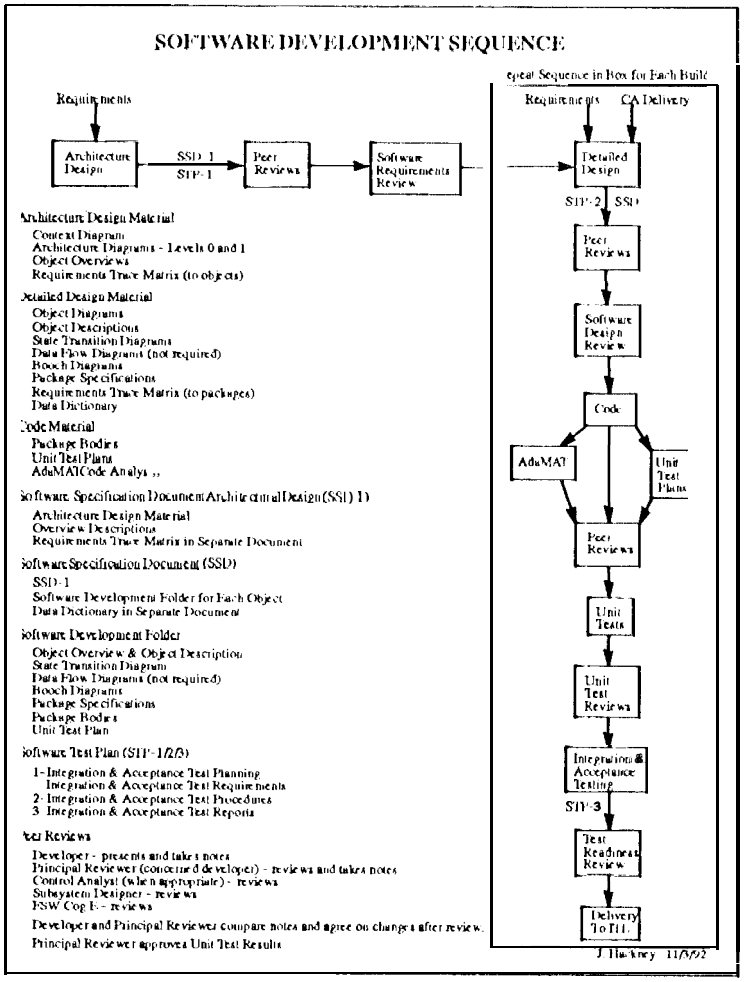

Fig. 6

opment tools wc had selected for the project.

It is important to note that, although the complexity of the software is very high, this is a fairly small software project in terms of the lines of executable code (estimated at Icss than 10,000). As will be seen, wc have identified just 30 objects and found it unnecessary to usc concepts like classes and inheritance. We could have used classes in the definition of the 1 lardware Manager objects, but wc found each to be sufficiently unique and the number to be so small that defining each separately better suited our process. This is especially true since, as is discussed later, wc identified the low level objects first, then identified the higher level objects in the architectural hierarchy.

Flight Software Context

The requirements placed on the Flight Software were described and illustrated in previous sections. The process of performing requirements analysis and architectural design actually began with the creation of a Context Diagram. The Cassini AACS Flight Software context is depicted in Fig. 7. A simplified view of the Flight Software is that inputs from the spacecraft sensors are used to compare desired attitude to estimated attitude and corrections are made by outputting to the actuators.

As can be seen in Fig. 7, the Flight Software has five primary interfaces:

1. The 1750A Computer and its operating System.
2. The Command and Data Subsystem (CDS) via the CDS Bus.
3. The Stellar Reference Unit (SRU) via the Pixel Interface and the AACS Bus.
4. Other AACS hardware via the AACS Bus including:
   a. The Sun Sensors - Sun Sensor Electronics (SSE)
   b. The Accelerometer - Accelerometer Electronics (ACCE)
   c. The Gyros - Inertial Reference Unit (IRU)
   d. The Thrusters - Valve Drive Electronics (VDE)
   e. The Engine Gimbals - Engine Gimbal Assembly (EGA)
   f. The Reaction Wheels - Reaction Wheel Assembly (RWA)
5. AACS Support Equipment (SE), for testing, via the Direct Access Unit (DAU) which is disabled in flight.

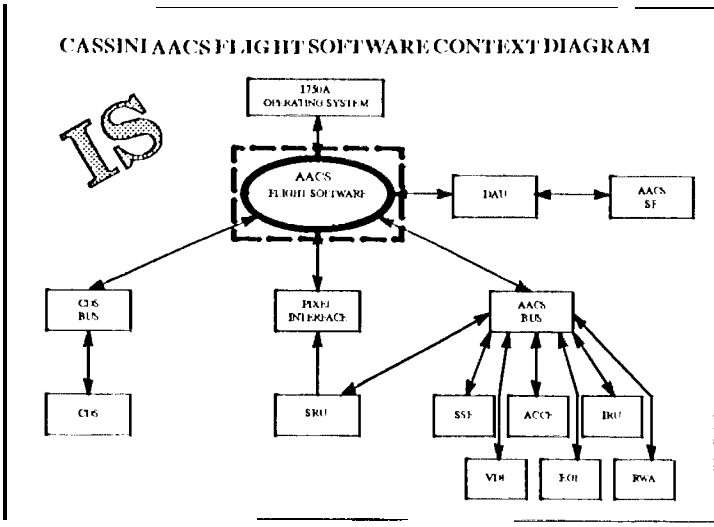

Fig. 7

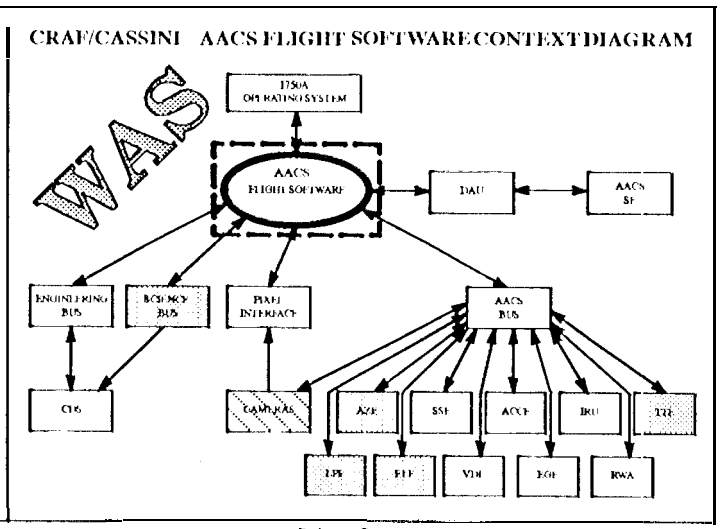

Fig. 8

Fig. 7 is a classical Software Context Diagram showing the software as a bubble, lines and arrows indicating data flow and boxes as terminators (external interfaces). We have taken the drawing a step further to show the next level of interfaces beyond the buses. This proves very useful since each of these interfaces (as well as the buses themselves) need to be represented in the software. The contents inside the heavy dashed line are expanded in the next level diagram.

Fig. 8 depicts the software context prior to the deletion of CRAF and the Cassini redesign. The shaded boxes were deleted and the Cameras were replaced by the SRU for Star Tracking. Accom-o-

dating these changes in the software design was straightforward and relatively painless because of the object Oriented approach as can be seen by comparing the "IS''t'WAS" Context and Architecture Diagrams.

Selection of Objects

After preparing the context diagram based on the subsystem requirements and subs ystem hardware architecture, subsystem Software Requirements Analysis and Architecture Design normally proceeds by first creating a strawman architecture of candidate objects, In our case, much of the control software architecture was dccidcd on during the Methodology Prototype. That design was extrapolated to the design of the Flight Software by the Control Analysts and is reflected in the Level 1 Architecture Diagram. in parallel, however, wc were doing a Context Diagram and brainstorming the objects nccdcd for the Level O Architecture Diagram. The object selection techniques described below were derived from the work of Don Firesmith and were considered for our approach.]n the final analysis, wc used four of the six techniques.

1. Terminator on a Context Diagram - each box on the Context Diagram became an object. Software units are almost always required to service each of the external interfaces. (Second oldest approach according to Ref. 3).

2. Abbotts' Noun Approach - look for nouns in the software requirements to consider as objects (e.g., The Flight Software shall use Sun Sensors for Attitude Estimation.) . in the example, both underlined nouns were converted to objects, but this technique was actually used to refine the architecture later in the process after software requirements were more mature. (In the real world, software requirements don't just fall from above, but must be laboriously derived in parallel with the architecture development). oldest technique according to Ref. 3.

3. Data Store on a Data Flow Diagram - processes which act on the data in a store are grouped with the store to become an object. We have not used data flow diagrams; therefore, this approach was not used.

4. Entity on an Entity Relationship Diagram (or nodes on a Semantic Net) - examples of each type of diagram are provided in Figs, 9 and 10, the similarity to Context Diagrams is apparent. Wc opted not to usc either of these diagrams and, therefore, didnt usc this technique.

5. Unit of Work - decomposing the software based on task assignments, Several objects were selected based on the convenient assignment of tasks both to the Control Analysts and to the Software Engineers. This technique can simplify and minimize both the human and the software interfaces.

6. Object Abstraction - is essentially a way of thinking in object oriented terms and requires a mind set (or paradigm) shift. After working with objects for a time (about six months) and thinking in terms of object size, minimizing coupling, and maximizing cohesion, onc begins to recognize obvious object candidates without using the other techniques. However, this technique is not clearly defined and is, therefore, not onc used by beginners.

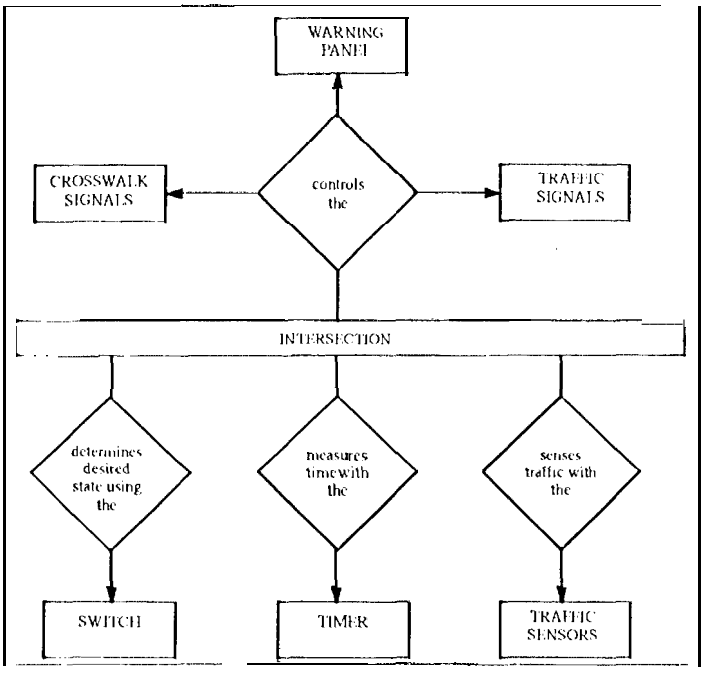

Fig. 9 Example Entity Relationship Diagram

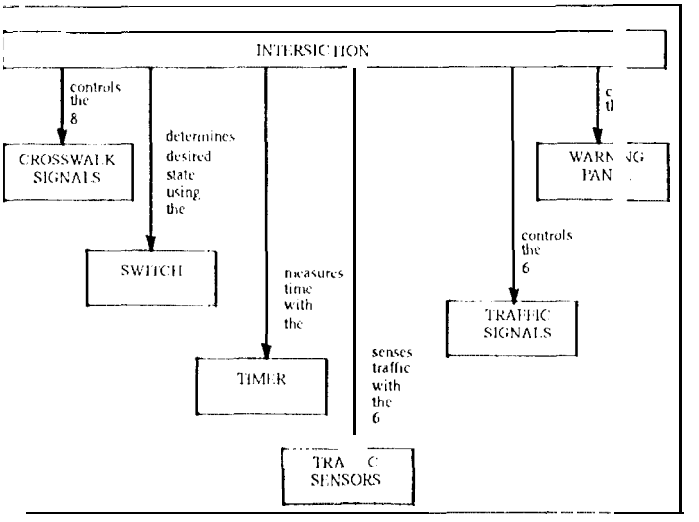

Fig. 10 Example Semantic Net

Grouping and Decomposing Objects

Parts of the Object Abstraction approach discussed above are the Classification Diagrams and Composition Diagrams (see examples in Figs. 11 and 12). We used the idea of (;ompositc' objects, objects

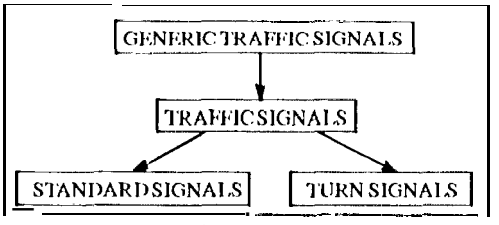

Fig. 11 Classification Diagram

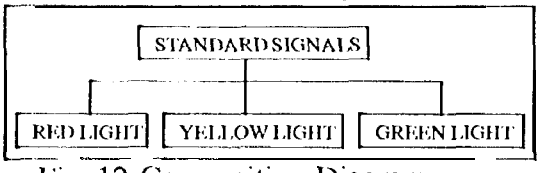

Fig. 12 Composition Diagram

composed of several, more primitive objects, to enable diagraming our software architecture in increasing levels of detail and complexity. We didn't use Composition Diagrams because our software architecture is not that complex, there are only two levels of diagrams, and the expansion of composite objects to primitive objects is relatively obvious. We could have used classes of objects for some of the hardware (e.g., thrusters, but we are actually interfacing to sets of electronics which control all the thrusters and it made more sense. to have the object represent the sets of electronics).

Flight Software Architecture

Possibly the most useful way to present the methods we used for developing the software architecture, is to walk through an Architecture Diagram and discuss the salient points. Fig. 13 is a portrayal of our highest level (Level 0) of software architecture. Fig. 14 is a view of the architecture prior to deletion of CRAF and Cassini redesign. Rounded rectangles represent objects. The square rectangle represents an external interface (or terminator). The arrows indicate direction of dependency (who is dependent on whom). We used a construct of arrowheads only to indicate a source which could be dependent on all objects or a sink which all objects could depend on. Mainly, this just kept the diagram cleaner. This kind of diagram is based on Dependency Diagrams, which look almost identi-

calto a Semantic Net (see Fig.10) without the description on the arrows. Early in the design process, control (via calls) are not known and this kind of diagram helps to start establishing hierarchy.

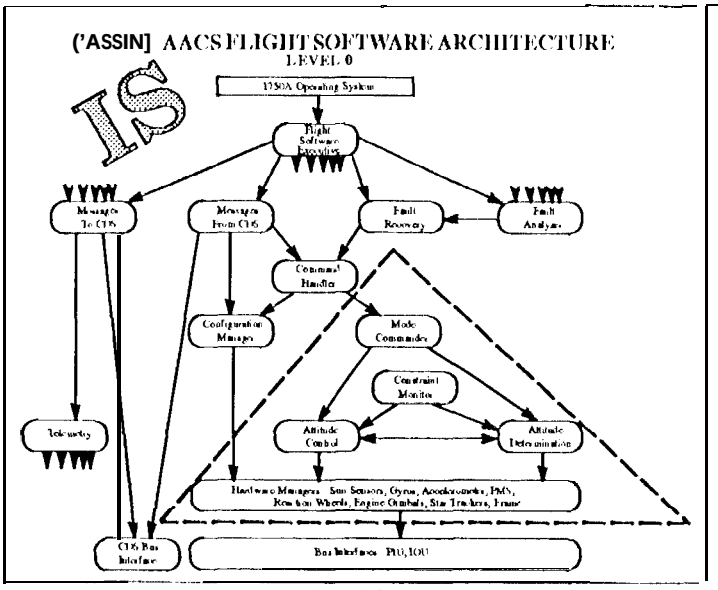

Fig. 13

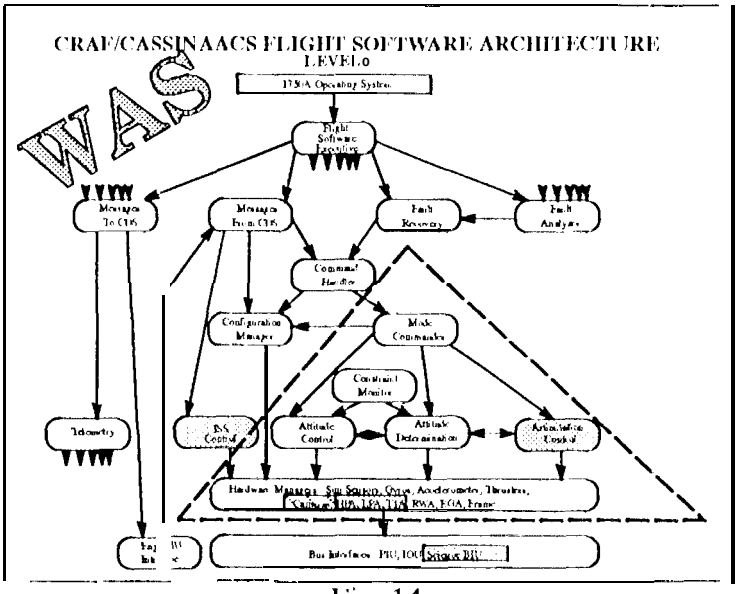

Fig. 14

Starting at the top, wc have the external operating system (the real time operating system, RTX, for the 1750A provided by '1'1 JD, our cross compiler vendor). The operating system links to the flight software executive as a result of an interrupt or during system startup.

The Flight Software Executive object services interrupts, handles timing services, schedules all execut ion and performs tasking. It is the only interface to the 1750A (via the operating system). The Flight Software Executive initiates a procedure in each object in turn when it is scheduled to execute.

Messages to the Command and Data Subsystem (CDS) may be created by any of the other objects but certainly by Telemetry, which can fetch telemetry data from any other object. Message packets are created and sent to the CDS (and perhaps on to storage in Solid State Memory or to the ground system) by initiating the CDS Bus Interface object which manages the bus protocol to the CDS (spacecraft) Bus.

Messages from CDS are periodically picked up from the CDS Bus Interface object and checked for validity. The Messages from CDS object then notifies the Command Handler that it has a command to process.

Faults may be raised and sent to the Fault Analyzer. Fault Analyzer will determine whether to kick off a Fault Recovery or just record the fault. Fault Recovery may involve a complex set of commands be issued which take priority over any commands from CDS.

The Command Handler determines which commands have priority, in the case of a conflict, whether the command is valid in the current software mode, and what actions are necessary to accomplish a command. Commands may be passed to the hardware Configuration Manager or the Mode Commander (software configuration manager) for execution.

The Configuration Manager maintains the status of the AACS hardware. Status such as In Use, Powered On, Ready, and Failed are kept for each device as well as the addressing path (e.g., 10 Bus A, IOU B, Electronics A).

The Mode Commander maintains the software configuration and allowable changes in the software configuration (mode changes). See Fig. 5 for the modes and allowable transitions. The Mode Commander sets goals for Attitude Control and manages the states of Attitude Determination.

Attitude Control is a composite object comprised of the Attitude Commander, Delta V (velocity) Control, Attitude Controller, and Inertial Vector Propagator. Plans for control, how the plans are to be carried out, and plan implementation (via Hardware Managers) are all done by the Attitude Control. See Fig. 15 for the expanded detail. Fig. 15 portrays the next level (Level 1) of architecture and at this level we have shown the actual control (via calls) and added the direction of data flow in the form of small arrows next to the control arrows.

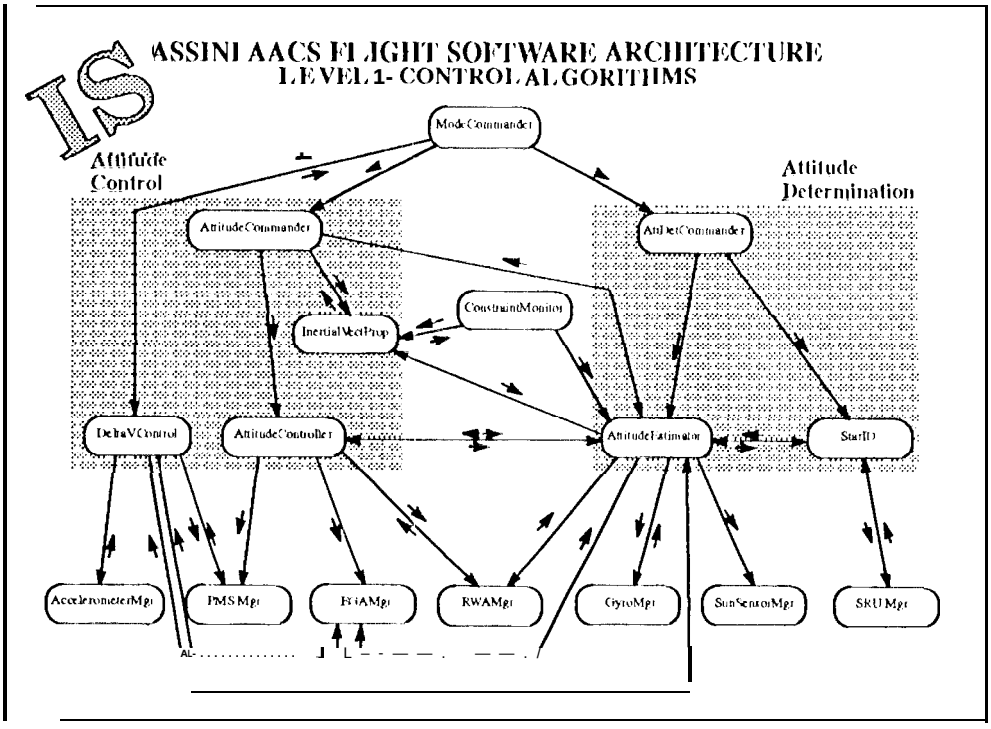

Fig. 15

Attitude Determination is a composite object comprised of the Attitude Determination Commander, the Attitude Estimator, and the Star ID (Identifier). Attitude Determination produces an estimate of current attitude, compares the attitude with the desired attitude and produces an attitude error by exchanging information with Attitude Control (attitude goals are achieved by this process of cycling between Attitude Determination and Attitude Control). Attitude Determination bases its estimates on data received from the hardware sensors (via the Hardware Managers).

The Constraint Monitor constantly checks both the estimated and desired attitudes for violations of forbidden spacecraft attitudes such as staring at the sun with the cameras or prolonged exposure of the instrument radiators to the sun.

Hardware Managers is a composite object and its expanded detail can be seen in Fig. 15. Each manager knows how to command a specific hardware unit and creates command packets for transmission by the AACS Bus Manager. A Hardware Manager also performs data conversion and compensation.

There are three bus interface managers, shown as the CDS Bus Interface and the composite object - Bus Interfaces: PIU, IOU. The Pixel Interface is via the Pixel Interface Unit (PIU). The AACS Bus Interface is via the Bus Controller Input/Output Unit (IOU).

The AACS Bus Manager takes packets from the hardware managers and prepares transmission packets for the bus. 11 handles putting packets in and taking packets out of the memory shared with the Bus Controller as well as servicing the handshake interrupts. Reply packets from the hardware are distributed to the individual hardware managers.

The Pixel Interface Manager sets up the Pixel Interface Unit (PIU) with an address where pixel data will be stored, enables the PIU to write to memory, and services transmission error interrupts. (Although the Star Tracker is commanded via the AACS Bus, output data from the Star Tracker is sent to memory via the Pixel interface).

The CDS Bus Manager puts transmission packets in the memory shared with the Bus Interface Unit hardware ready for pickup by the CDS and removes transmission packets sent by CDS from the shared memory. It sets up and maintains the. protocol to the Bus Interface Unit and handles the handshake interrupts.

All of the objects are capable of detecting errors or faults which are specific to their own specialized knowledge and raising those faults to the Fault Analyzer. All of the objects also generate telemetry data which is picked up by Telemetry.

Fig. 16 shows the Level 1 architecture prior to deletion of CRAF and Cassini redesign. Note that Articulation Control has been removed (only Engine Gimbal articulation remains and is now part of Attitude Control).

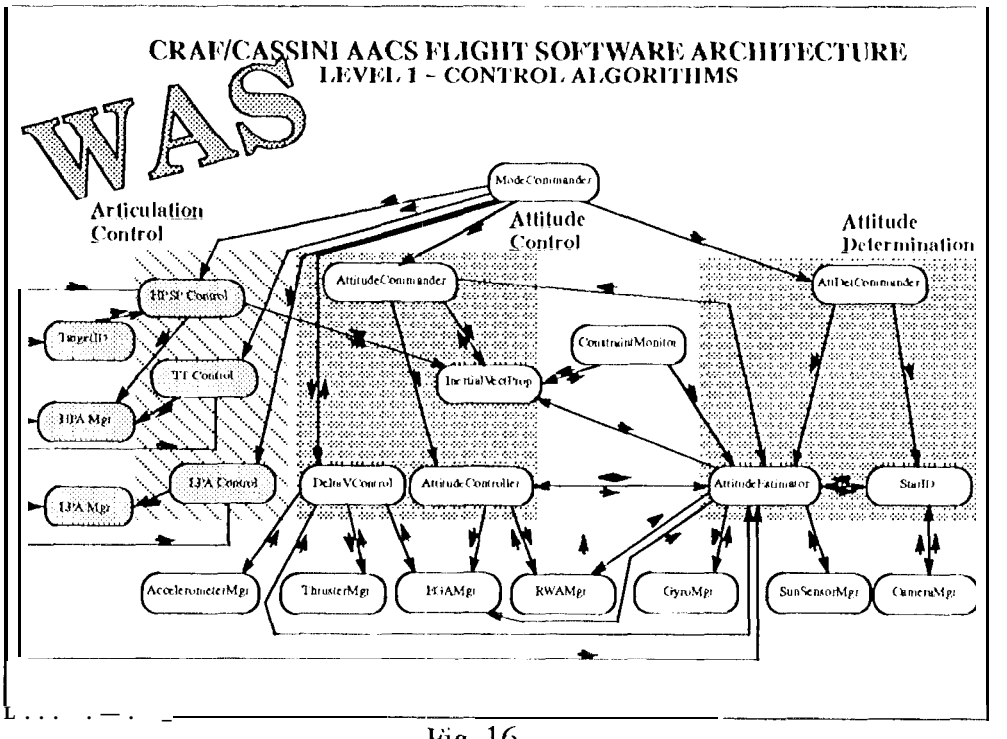

Fig. 16

The following table lists the criteria used for each of the objects from the six approaches previously described. Expanded explanations are as follows:

Context, Noun: originally derived from the Context Diagram, later included in requirements.

Noun, Work Unit: originally derived from experience, including prototyping, as to what work units are most useful, later included in requirements.

Work Unit, Abstraction - brainstorming: originally identified during our architecture brainstorming meetings, later verified as a useful work unit.

Context, Work Unit: originally derived from the Context Diagram, later verified as a useful work unit.

Noun, Work Unit: originally identified as a useful work unit, later included in requirements.

Noun, Abstraction - brainstorming: originally identified during our architecture brainstorming meetings, later included in requirements.

OBJECT	Selection Criteria	Explanation
1. Accelerometer Manager	1 & 2	Context, Noun
2. Attitude Commander	2 & 5	Noun, Work unit
3. Attitude Controller	2 & 5	Noun, Work unit
4. Attitude. Determination Commander	2 & 5	Noun, Work unit
5. Attitude Estimator	2 & 5	Noun, Work unit
6.11111 Manager	1 & 2	Context, Noun
7. Command Handler	5 & 6	Work unit, Abstraction - brainstorming
8. Configuration Manager	5 & 6	Work unit, Abstraction - brainstorming
9. Constraint Monitor	5 & 6	Work unit, Abstraction - brainstorming
10. Delta V Control	2 & 5	Noun, Work unit
11. FGA Manager	1 & 2	Context, Noun
12, Fault Analyzer	5 & 6	Work unit, Abstraction - brainstorming
13. Fault Recovery	5 & 6	Work unit, Abstraction - brainstorming
14. Flight Software Executive	1 & 5	Context, Work unit
15. Frame Manager	5 & 6	Work unit, Abstraction - brainstorming
16. Gyro Manager	1 & 2	Context, Noun
17. Inertial Vector Propagator	2 & 5	Noun, Work unit
18, IOU Manager	1 & 2	Context, Noun
19. Messages From CDS	1 & 5	Context, Work unit
20. Messages To CDS	1 & 5	Context, Work unit
2 J. Mode Commander	5 & 6	Work unit, Abstraction - brainstorming
22. PIU Manager	1 & 2	Context , Noun
23. PMS Manager	1 & 2	Context, Noun
24. PROM Control	5 & 6	Work unit, Abstraction - brainstorming
25. RWA Manager	1 & 2	Context, Noun
26. SRU Manager	1 & 2	Context, Noun
27. Star ID	2 & 5	Noun, Work unit
28. Sun Sensor Manager	1 & 2	Context, Noun
29. Telemetry Manager	2 & 6	Noun, Abstraction - brainstorming
30. Utility (Global) Functions	5 & 6	Work unit, Abstraction - brainstorming

Design Representation

The detail of each object's design is portrayed with at least an Object Diagram, a State Transition Diagram, and a Booch Diagram. The Object Diagram adds detail to the Level 0 and 1 Architecture Diagrams in the form of named events and data which pass between the object of focus and other

objects connected to it. Adding this detail in a higher level of diagram would be impractical for drawing purposes, plus a reader can track through the design from higher to lower levels and see an increasing amount of detail. Fig. 17, an example Object Diagram from the second prototype, is a good way of showing the ideas without the complexity of the full blown application.

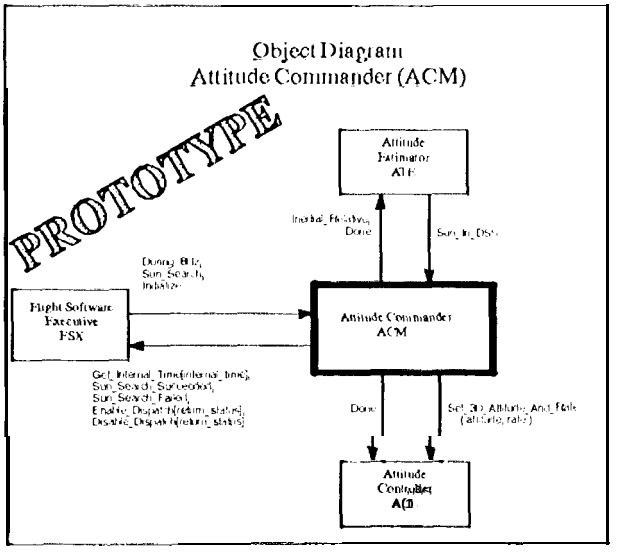

Fig. 17

State Transition Diagrams were very useful during the Requirements brainstorming phase. The State Transition Diagram portrays the internals of an object - its states, events which cause transitions between states, and the processing which occurs during a transition and within a state. Concurrency can also be portrayed as can be seen in Fig. 18, an example State Transition Diagram (or Statechart) from the second prototype, where a process which returns attitude and rate runs concurrently with the process which computes the estimated attitude and rate (concurrency is indicated by the dashed line running through the large superstate).

The hooch Diagram portrays the actual implementation packaging of the design in the form of Ada Packages. The packages with the heavy borders are the packages which make up the object of focus. Dependencies on the packages of other objects, either from the Package Spec or Package Body (indicated by the B or S), are represented by arrows and the packages themselves, Only the external packages referenced (via Ada *with*) are drawn, not all the packages of a referenced object, This promotes tracking and diagraming. In the example Booth Diagram from the second prototype, Fig. 19, notice that visible procedures and types are shown (but not all are shown, we opted not to show the global procedures and types - math routines and types used by three or more objects - as they are too voluminous). Visible data can also be portrayed.

Any additional drawings (e.g., Data Flow Diagrams) which aided the developer in understanding the problem or describing it to others were encouraged but not required.

During the Prototype, we created models of our deliverable products (i. e., documents) and guidelines for our development procedures (Design, Code, Test, Configuration Control, Problem Reporting, Reviews). The guidelines include templates for the textual material (see Fig. 20), diagrams, and code format.

Tool Usage

We found that tools in the form of diagrams were very useful in documenting the results of our analysis and in providing the media used in brainstorming how things should work.

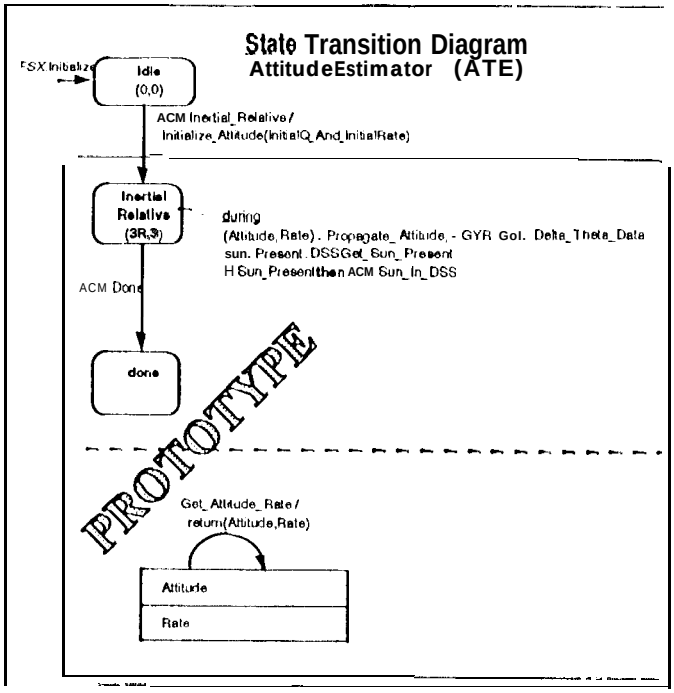

Fig. 18

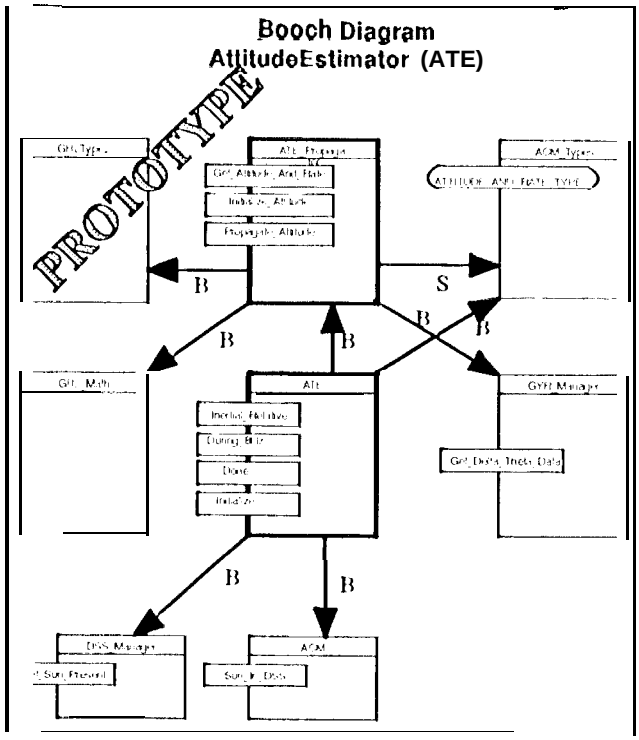

Fig. 19

Object Overview
Object Name. (Object Acronym)

1. **Description:** <In this section describe "what" functions this object performs, NOT "how" it is done.>

   <This is the component for a continuation paragraph>

2. **Inputs:**

From	<Object Acronym>	<Actual Event/Function Call as described in the Object Diagram>
Example		
From	*ATE*	*Get_Attitude_And_Rate[Attitude Quaternion, Rate Vector]*
	FSX	*Initialize*
	ACM	*Done*

3. **Processing:** <In this section describe "how" the. functions described in the description section are performed>

4. **Outputs:**

To	<Object Acronym>	<Actual Event/Function Call as described in the Object Diagram>
Example		
To:	*THR*	*RCS_Thruster(Thruster_Vector)*
	ATE	*Done*

Fig. 20 Textual Templates

Object Description
Object Name (Object Acronym)

Description:

<Description goes here.>

Public Operations:

Example

IBL_Math.Conjugate

Public Attributes:

Example

IBL_Constants.Kilo

Standard Operations:

Example list

IBL_Math.Initialize

description:	*Clears all accumulators to the standard state*
assumptions:	*must be executed before any other routine in this package is called*
inputs:	*none*
processing:	*all storage in the package is set to zeroes.*
outputs:	*none*

Operations:

Example list

IBL_Math."" Binary operation (overloaded)*

description:	*VECTOR32_4_TYPE x VECTOR_32_4_TYPE -> VECTOR32_4_TYPE; subtracts the arguments component by component*
assumptions:	*none*
inputs:	*any value*
processing:	*constructs a new value of the same type by component-by-component subtraction*
outputs:	*returns the constructed value*

Integrated automated development tools to help in creating our drawings and maintaining the dictionary of entities in the drawings appear to be too immature for object oriented development, too inflexible to be able to adapt to our particular approach, and overkill for a project of this size. We, therefore, used either MacDraw or Interleaf for our diagrams and Interleaf for producing our textual material and publishing our documents. Any MacDraw diagrams were filtered and imported to Interleaf when the documents were created. The Statemate tool from 1 logix comes the closest to doing our statecharts (State Transition Diagrams) and might be looked at both for methodology and tool support (wc no longer had funds for tools of this scale when wc realized the possible benefits).

Wc are considering a Data Base tool for our Data Dictionary and Requirements Trace Matrix. Wc are currently doing the Requirements Trace Matrix on Interleaf.

LESSONS LEARNED / SUMMARY

Prototyping all the people interfaces, software products, and development processes that schedule and budget allows, is time and effort well spent.

Don't underestimate the need for a protot ypc effort to work through the ramificat ions of an Object Oriented approach and get experience with the process. Grouping data with the only procedures which are allowed to act on that data seems like a simple concept but the resulting changes in the software development paradigm are abundant, Although we actually did two prototypes, the first was dedicated to deciding on the development paradigm (Object Oriented obvious] y won) and working through that paradigm with all the groups involved in the software development, not just the 1 'light Soft ware team. The second protot ypc was an extension of the first where the I 'light Software team concentrated on defining, refining, and gaining experience with the details of the process. Remember too that the prototype is a very uscf ul tool for wringing out test bed simulations,

State Transition Diagrams are invaluable as a tool for driving out both requirements and design details. Wc highly recommend the USC, development and maintenance of these diagrams during the entire soft ware development process. A pictoral representat ion of what needs to occur in a system and what can cause those things to occur, aids the discussion, understanding, and transmission of requirements and design details.

That *data in an object may only be changed by the object (no data is shared in common)* is the most important premise of an Object Oriented soft ware development approach. Objects may be chosen in several ways, just try to make the interfaces as simple and few as feasible and place data and processes in the objects that have the most knowledge about them. The independent objects, which result from these few simple premises, accomodate even major system changes and can be reused.

With all the up front preparation, it might be difficult to separate the contribution of the Object Oriented paradigm from the contributions of the other newly adopted tools and techiques, were it not for the significant redesign of Cassini and deletion of CRAF. Software development proceeded without significant impact or interruption during all that turmoil and this can be largely attributed to the object oriented architecture.

Our experiences to date indicate that the Object Oriented approach to developing flight control software is very promising. Wc will continue to assess the process through the implementat ion and test phases.

ACKNOWLEDGEMENTS

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

The spacecraft configuration drawings were produced by H. W. Price and the original Level 1 Architecture Diagram was drawn by E. C. Wong.

The authors would like to thank the following for their substantial contribution to the development of the methodology: M. A. Bramble, J. Y. Lai, G. K. Man and E. C. Wong.

REFERENCES

1. David Harel, "Statecharts: A Visual Formalism for Complex Systems," Science of Computer Programming 8 (1987) pgs. 231-274.
2. Sally Shlaer and Stephen J. Mellor, "Object oriented Systems Analysis: State and Process Models," class from Project technology, Inc.
3. Don Firesmith, "Object-Oriented Requirements Analysis," TRI-Ada '90 Tutorials Volume 1 pgs. 3-1 through 3-172.